BUILDING DATABASE SYSTEMS

DATABASES

DESIGNING A DATABASE

Database applications are everywhere, making up more than half of the work processed by computers. Banks, insurers, tour operators, hoteliers, telephone directory enquiries and even public libraries are among the places you'll encounter database systems.

A database is a very effective way of organising data records so that users can access information quickly, add new data and change existing records. An effective database strategy must make searching for specific data in a mass of information a short and simple process.

The same strategy will be effective for many smaller database applications, such as keeping a list of club members, an address book, or a catalogue of Compact Disks and tapes.

You need two skills to create a working database system from scratch. Programming knowledge must be combined with the know-how to design database structures. For this reason we

STARTING OFF

This series is split into two sections, running in parallel. In *Designing a Database* we look at the basic principles that apply to designing any database project, regardless of which database package or programming language you use. In *Developing with SR-Info* we apply these principles to a project using *SR-Info* – the language featured on the issue 54 *SuperDisk*. You will need to have installed *SR-Info* in order to follow our project.

DEVELOPING WITH SR-INFO

On this month's *SuperDisk* there are files (in the MAGAZINE directory) containing changes to the *SR-Info* database program that we are developing. The changes incorporate commands and structures that we have been studying and bring one function closer to completion – the LIST CHOSEN ITEM selection from the menu in the ARTILIST program.

Around line 18 there's a change to the DO CASE structure:

```
CASE MCHOICE=3
     DO CASE
     CASE IDXA="**"
     DO LISTITEA
     CASE IDXS="**"
     DO LISTITES
     CASE IDXT="**"
     DO LISTITET
     ENDCASE
```

This introduces a new DO CASE inside the existing one – the new structure links (with the DO command) to a different program, depending on which of three fields (IDXA, IDXS or IDXT) has asterisks in it.

An inspection of the rest of the program will reveal what these fields mean – they are the displayed tags that show which sequence is being used. Effectively they give an answer to the question, 'which is the active index?'.

A shortcoming of database languages generally, is that it's not easy for a program to 'know' certain things, like the current active file, the selected slot, or the active index. However, it is easy to set the information in a field (or multiple fields) as we have here.

We originally planned to have one program, LISTITEM, to perform this function. But as we analyse the task, we see that our project really calls for three separate programs. If we are in AUTHOR sequence, the function should ask for a specific author; if in SUBJECT sequence, a specific subject; and if in TITLE sequence, a specific title.

The first reason why it's a good idea to divide this task into three is that the tasks are clearly separate (those identified by IDXA, IDXS and IDXT in the code above). We can then pass any required information from the higher level calling program to a sub-program (one that we DO from the base program), but remember that the fields have to be deemed GLOBAL for the sub-programs to know about them.

The second reason is that the decision to search for and display information made in one program (LISTITEM) would involve coding in several places. In the new scheme, the decision is made once in a simple DO CASE structure.

The only argument against having three programs like this is that the resulting full program (when finally compiled) will be longer. You must use your judgment as to whether this is a decisive argument. In moderately-sized programs (certainly in this one) this should not be very important.

The decision to produce three sub-programs results in three similar sections of code. If we look at one of these sub-programs in detail, we will know all but minor details of the other two. We will look at LISTITEA:

```
WINDOW 10,10,20,70
CLS
SET EXACT OFF
SET TALK OFF
FINDER="[25 spaces]"
```

Most programs (even sub-programs like this one) will begin with some housekeeping instructions. In this case, we first

have built this series in two distinct parts. In this part we will look at the design principles.

INDEXES – THE QUICK WAY TO A DATABASE'S HEART

At heart, database systems store information in elements called 'files', 'records' and 'fields'. Each field contains a piece of information (such as surname); each record is made up of several fields. The database file is a collection of records with the same structure (that is, made up of the same types of fields).

The database records can be accessed in physical sequence, but this is not very helpful for two reasons. Firstly, it's rather time-consuming to search for a particular record one-by-one through the whole database. Secondly, any new records will just be added on to the end – so even if there had been some sense to the sequence when the database was built, it would be lost as soon as additions were made.

PATHWAYS THROUGH THE DATABASE

To overcome these difficulties, 'index files' are built, defining pathways through the database file in each sequence needed by users. Index files do not carry any fields – only pointers to records, like the page number references in the index of a book. It is quite common to have more than one index to a database, so that there are several sequences in which the records of the database can be read (although it's interesting to note that in some database implementations indexes are not shown as distinct files, but look as if they are part of the main database file).

In each index, the sequence of the records is determined by the 'key', which is usually made up of one or more fields appearing in the record structure. Each index has its own separate key, and therefore its own specific order. One index may steer a path through the database in 'surname' order, where another goes through in 'town'/'surname' order.

Using an index we can take shortcuts when looking for a particular record. Although this process is complicated for the database manager software, all that the programmer or user has to do is specify a 'key value', and the first record to match it is found almost instantly.

So, as well as allowing us to read through the database in a particular 'key sequence', the index also lets us go directly to the record we want. The database designer must pay special attention to what indexes are available for each database file, and how their keys are constructed.

BUILDING MANY INDEXES

When you want access to a specific record in a database you must use the correct index.

For example, if you wish to look for a record relating to Mr Smith, you could use the 'surname' index, searching for a key with the value SMITH. But it would be faster and wiser to use his account number (if known) as the appropriate index. Why? – because there will be only one record with that particular key value, so we would find the right Mr

BUYING TIME

***SR-Info* is a product of Sub Rosa Inc of Downsview, Ontario, Canada, and is marketed in this country by New Star Software, on (0245) 265017.**

***SR-Info* is shareware. If you continue to make use of it after this series is over, or put it to use in a commercial application, you should register your copy with New Star. The company will then send you the latest version, along with a comprehensive manual.**

define the window, or screen portion in which we will be displaying. CLS then clears the window.

SET EXACT OFF is something you would add only after coding the rest of the program. It controls whether or not partial keys can be used. This will be demonstrated later.

SET TALK OFF prevents the display of messages more helpful in *SR-Info's* conversational mode (when you are not in the middle of a program).

The final line of housekeeping we have is an equate to FINDER – our own local variable. This brings it into existence and also establishes its size at 25 characters.

```
@ 10,10 SAY "AUTHOR.." GET FINDER
READ
FINDER=TRIM(FINDER)
FIND &FINDER
```

These few lines are the germ of the program itself. The '@ SAY GET' structure is a powerful way of communicating through the screen and keyboard. The '@' is followed by a space, then the row and column number for what follows. SAY puts a display onto the screen at the specified location, and GET displays its current value and waits for a new value to be entered.

In practice, a whole group of GET commands are written one after the other, but none will be started until a READ is executed. READ initiates keyboard input into the FINDER field.

TRIM is an example of a function. Functions can be recognised by the parentheses that follow them. TRIM(FINDER) says 'look at the field FINDER, trimming off all trailing spaces'. Preceding this with 'FINDER=' tells the function where to place its result. This little trick ensures that FINDER is reduced to its significant characters. This is essential for the partial key search, as we will see. Before the TRIM function, FINDER is a 25-character field, even if only the letters ABC were keyed into it. After the TRIM, FINDER will be shortened to the desired length.

FIND performs the whole of the search operation on the active database file, using the current index. Because we have SET EXACT OFF, this command will do a partial-key search; if SMI were keyed into FINDER, SMITH would be found. Note that the fieldname FINDER is preceded by an ampersand [&]. If we had coded FIND FINDER alone, the author's name being sought would be 'FINDER' – certainly not our intention.

```
IF #=0
? "NOT FOUND"
? "Press any key to continue"
WAIT
CLS
WINDOW
RETURN
ENDIF
```

This bit of code handles the 'not found' condition for our search. The crosshatch [#] alone means 'the current record number', using the natural sequence number within the database file. If the FIND doesn't fetch a matching record, the sequence number is zero. The question mark [?] is used as an output command – here it means 'output to screen'.

At this point, note (by executing the program) that the question mark command is restricted to the defined window. By contrast, @ SAY prints to the screen in a particular place, ignoring the window limits. It never causes scrolling, but overwrites any information left on the screen in that position.

WAIT works rather like the MS-DOS command PAUSE. It waits for a keystroke to unfreeze the program. When that keystroke is made, the screen is cleared (CLS), the full screen is opened again (WINDOW with no operands), and the program is

Smith first time.

In fact, this question of 'unique' versus 'non-unique' keys is very important. Indexing on a non-unique key is fine for re-sequencing your records – an index built on surnames would let you print out an alphabetical list of people, for example. However, if you want to access individual records directly in an accounting application, for instance – then you'll need to build indexes where each record has a unique key value.

In general, you would construct several different indexes through a single database, so long as each is useful in searching for particular records. In an on-line enquiry system at your local library, for example, you may wish to locate a book whose title you know; on another occasion you may not have a title, but know the author; or you may need a list of books on a particular subject. Each of these searches can be made easily, as long as the database has a suitable index for each search. Here are some of the key questions to ask yourself when designing the index files to go with a database:

1. In which sequence do I want the reports to be made?

(The answers to this question will dictate what indexes must be constructed)

2. What distinctive details will sometimes be available when enquiry into the database is needed? Each detail will require a separate index, with that detail field as part of its key. If there is some detail that will always be present, it should be added as the first item of each key.

3. What major details, taken together, would usually be enough to identify precisely one record in the database? The answer to this question will warn you if some of your indexes have keys with unneeded fields attached; for instance, the DEALER NAME field in the structure of a DEALERS database is probably not a good thing to have in an index – even if (by rule 2) it is available when an enquiry is made – because having this value will not really help you be 'more specific' about which record you want.

MORE THAN ONE DATABASE ?

Last month we introduced the concept of related database files. Further savings in size (though at a slight cost in access time) can be realised by noting fields within the original database structure that could be abbreviated – even if the abbreviation looks meaningless.

In our project we supplied a field called MAGAZINE where we record the abbreviated name of the magazine in which each article has appeared. If we had recorded the whole magazine name in the primary database, the field length would probably have to be 25 columns wide. However, the way we have set things up there are only three columns for the abbreviation (PCP means PC PLUS; NCE is the shortened form of New Computer Express, and so on). A related database (called MAGFILE) uses this abbreviated field as its key. When the record matching the key is brought into memory, we see it has the full name, PC PLUS – and its address, editor, frequency and so on.

Many database systems provide automatic linkage of database files in this way, primarily to give expansions of abbreviations and further details – it's this feature that makes such a system a relational database.

When related databases are linked by keys, the process involved in getting a record into computer memory will

finished (RETURN). ARTILIST is re-activated after the DO LISTITEA command – but there is nothing else to do within the DO CASE structure.

```
DO WHILE AUTHOR=FINDER.AND..NOT.EOF()
@ 11,10 SAY TRIM(FULLNAME#2) + " " + STR(ISSUE,3)
@ 12,10 SAY "page "+STR(PAGE,3)
@ 14,10 SAY '"' + TRIM(TITLE) + '"'
@ 15,10 SAY "Subject.. "+SUBJECT
? "Press any key to continue"
WAIT
SKIP
ENDDO
CLS
WINDOW
```

This DO WHILE structure will be executed repeatedly, while the condition still exists; the condition is that the AUTHOR for each database record accessed is a match for FINDER (which the user typed in earlier), and that we have not reached the end of the database file.

We test for this with EOF(), a function whose value is true when the end of file has been reached. Note that both the words AND and NOT must have their own fullstops before and after them – this means two fullstops between them.

You have seen TRIM, and you know that PAGE, TITLE and SUBJECT are fields within the record structure of the ARTICLES database – our active file. But what is FULLNAME#2 ? FULLNAME is the name of a field in the MAGFILE database; the #2 (which must be immediately after the fieldname, without any space intervening) specifies that the database being used has been assigned to the SELECT slot two.

We built up the relation between ARTICLES and MAGFILE back in ARTILIST.PRG, in the following lines, (taken from the TOTRECS procedure)

```
USE#2 MAGFILE INDEX MAGREF
SET RELATION ON MAGAZINE TO 2
```

Once that has been set, every time we change which ARTICLES record is in memory, the program will automatically adjust the MAGFILE so that its matching record comes into memory. We saw that we found the first desired record with the FIND command – but where do the subsequent ones come from?

The answer is SKIP – a simple command which means, 'change to the next record in the database, going along the current index pathway'. If there is no further record, the record number [#] is set to zero, and (as we have seen) the EOF() function would become true – and the DO WHILE loop would not be repeated any longer. After the ENDDO, CLS and WINDOW finish off the process. (You can always think of a RETURN being at the end of a program, even if it is not coded, so we again go back to the ARTILIST program).

ON YOUR OWN

Try ARTILIST now, paying attention to the changes in the 'LIST CHOSEN ITEM' part of the program. Feel free to alter the programs and databases in several ways. If you have retained the *SuperDisks*, it won't be difficult to undo damage.

There is an intentional bug left in the new programs, concerning the way data is displayed – can you find it, and (more to the point) can you correct it?

You now have nearly enough to code the 'LIST ALL ITEMS' part of the program; it should work in a similar way to 'LIST CHOSEN ITEM', with one big difference. Each record should be displayed, and a key press awaited; if [PgUP] is pressed, the previous record is displayed; if [End] is keyed, the function is finished; anything else will cause the display of the next record.

You will need some extra information. The INKEY() function, which waits for and reads a keystroke as a numeric value ([PgUP] is 329, and [END] is 335) is the first bit. The second bit is that where SKIP goes ahead one record in sequence, SKIP –1 goes backwards by one record. At the beginning of the file, the record number is set to zero, but EOF() is not set to true. Next month you can compare your results with ours.

WE COULDN'T HAVE SAID IT BETTER.

"could revolutionise DTP"
NEW COMPUTER EXPRESS - OCTOBER '90

"promises to set the budget DTP world smouldering"
PC PLUS - DECEMBER '90

Introducing PagePlus 1.0 from Serif, the intuitive new Desktop Publishing package for all Windows 3.0 users.

£85
INCLUDING P&P AND VAT

What Personal Computer thought PagePlus "could be a winner" while PC Plus enthused "there's no reason why this program shouldn't become the Windows 3 DTP for general use".

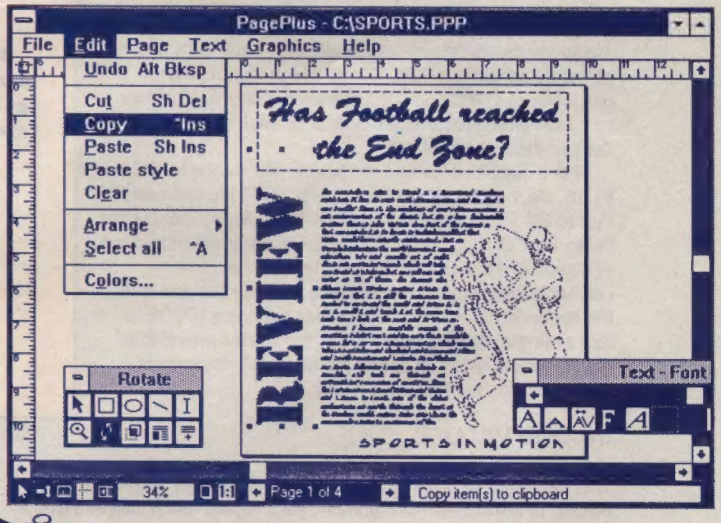
Described by PC Today as "perfect for short, single chapter documents" PagePlus is packed with features: multiple page views, rulers, guides and true type display. Variable type sizes in tenths of a point, and the ability to expand, condense, kern and rotate text. Sizing and cropping of pictures. Irregular text wrap. And the list goes on... "A surprisingly powerful DTP package" commented What Micro? magazine.

Despite such flexibility, PagePlus is refreshingly easy to use. Our unique panel is "wonderfully interactive" (What Personal Computer) and "makes layout easy" (New Computer Express).

PagePlus is truly comprehensive: text and graphics import (including TIFF and PCX); a PANTONE® Color License; 70 interactively scalable typefaces; a library of over 100 images to enhance your pages (use "as is" or customise in Paintbrush); and output to most Windows supported printers from dot matrix to PostScript. Simply "amazing value" concluded PC Plus.

Camera ready, colour separated artwork was produced using PagePlus 1.0 (and included clip art samples) with output to a QMS PS810 (PostScript) laser printer.

At only £85 (inclusive of postage and VAT!) PagePlus is the new price leader in DTP. And you even get a thirty day money back guarantee. Now you can see for yourself: simply phone or fax your order direct to Serif, or mail the coupon today. You'll need Windows 3.0 running on an AT compatible (286 or higher) with at least 1mb of memory (2mb or more is strongly recommended).



Name _____
Phone _____
Address _____
_____
_____
_____
Credit Card _____
Expiry Date _____
Signature _____
NB Please include card billing address, if different from above.

☐ Please send me a PagePlus information pack.

☐ Please send me PagePlus.
Please specify 5.25" (1.2mb) **OR** 3.5" (1.44mb) disks.
① I enclose a cheque, PO, bankers draft or international money order.
② Please bill my credit card £ **only.**
③ I enclose an official purchase order from a government department, local authority, educational establishment or plc.

Please mail to:
Serif (Europe) Limited
PO Box 15 West PDO
Nottingham NG7 2DA

Or, if you prefer, call or fax:
✆ (0602) 421502
Fax: (0602) 701022

PagePlus is £85 inclusive of P&P (£3.91) and VAT (£11.09) for UK (&NI) orders only. Anticipated delivery is 2 to 4 weeks: you will be notified of any delay. Next day delivery normally available for an additional £6.90 (inc. VAT). Orders are sent with VAT invoices. European orders are £80 (inc. £10 P&P; VAT not applicable). Generous site licenses available for business/education. Sorry, no discount or evaluation copies available.

involve the gathering of fields together from several different databases. When information is divided skillfully between several databases, duplication and waste of fields can be kept to a minimum, and changing information is greatly simplified. For example, if the address of a client's office changes, the database operator need only modify one record (in the 'client' database) rather than several (in the 'invoice' database).

Another benefit to this relational approach can be seen when new records are added to the primary database. If a wrong (or unused) abbreviation is used, a simple lookup procedure can confirm that the abbreviation has not been made available, and that there is no record to match it in a related database. Given this information, the operator can then either correct the error, or confirm the intention to make a new record for the related database as well.

GETTING ON WITH RELATIONS

It is easy to establish whether you should convert a database into two or more related database files. First, ask whether there are fields where the same information is repeated in different records. For example, the same magazine title, frequency and cover price probably appear several times through the ARTICLES database. In this case, those fields should form a new record structure in a separate database, with one record for each different magazine.

Secondly, see if the same fields appear in more than one database and have the same meaning. If so, you should find out whether there is a linking field, and establish a relation between the two databases, deleting the field from one database where it's unnecessary.

PUTTING A DATABASE THROUGH CHANGES

Among the many benefits of the database approach is the room for changes in the structure of the records. Often this can be done without disrupting the existing programs associated with the database, and without risk of losing existing information in the database.

However, any change of relationship between existing databases is more difficult. It will save you having to make more complicated changes if you anticipate possible relationships between database files at the start – even before the supporting databases exist. For example, MAGFILE (in our project) was not even fully planned when the ARTICLES database was constructed, but we allowed for its future existence without knowing what fields (editor, address, frequency, and so on) would be part of it.

As a general guideline, remember that new database files can be created, with simple relationships to the primary database and that index files, defining new pathways and relationships, can be created easily, and at little cost.

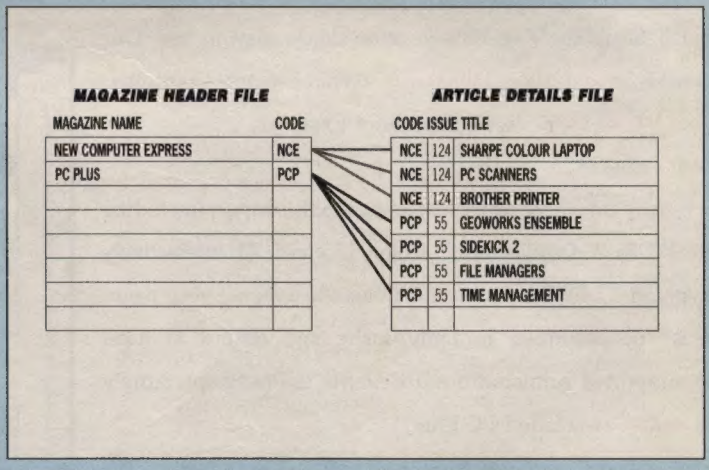
FURTHER CONSIDERATIONS

Many database systems allow partial keys, so that records can be matched with as much of the key as is available (starting from the left). If there is an index file with the key TOWN/SURNAME, but only TOWN is known, the index will work, finding the first record with matching town. Note that there is no need for an index in TOWN sequence if one with TOWN/SURNAME sequence is already in existence. ●

HOW RELATIONAL DATABASES WORK

In this simplified example we have two database files, 'magazine headers' and 'article details'. Each file's record structure contains a field, CODE, which identifies a particular magazine, and the contents of this field allows us to see which records are related to each other across the two database files.

For example, if we find the header file record for PC PLUS, we see that its CODE value is 'PCP'. We can now find all PC PLUS's records in the articles file by looking for those with 'PCP' in the CODE field. Similarly, when looking at a record in the articles file, we can find its corresponding header record by looking for the one with the same CODE value. So while the field name (CODE in this example) defines how the two databases are related, it's the actual data ('PCP' and so on) which determines which records are related to which.



UNIQUE VALUES

This diagram is a good example of 'unique' versus 'non-unique' key values. In the HEADERS database, each record has a unique value in its CODE field – there is just one header record per magazine, and the CODE value (e.g. 'PCP') must identify that record and no other.

In contrast, many records in the ARTICLES database can have the same CODE value as each other, a situation which mirrors real life. We would expect to have many articles all relating to the same magazine, so saying that an article 'belongs to PC PLUS' does not uniquely identify it from others – thus the value of an article record's CODE field is expected to be 'non-unique'.

We call this a 'one to many' relationship – one magazine header record relates to many article records. In such a relationship, the 'one' file will normally have a unique-value key field, and the 'many' file will have a corresponding non-unique-value key field, just as in our example.

As a database designer, you have to make sure that the key fields exist to support the relationships you want. Truly unique key values seldom occur naturally – surnames, names, addresses and the like are definitely not suitable – so you have to generate them artificially, which is why we have account numbers, codes and so on.

THE ROLE OF INDEXES

Our diagram shows the records in the articles file neatly sorted according to their CODE field values, but in practice the records in a database are never physically stored according to a 'key sequence'- they are stored according to the chronological sequence in which they were added to the file.

In strict 'relational theory', this problem of chronological sequence is overcome by searching serially through each database, looking for the required key values. In real life though, databases almost always use indexes to speed things up. An index enables the records in a database to be accessed as if they were physically stored in a key sequence.

In our example, the headers file would have an index based on its CODE field, as would the articles file. Having found a record in either file, it would then be possible to jump directly to the first (or only) corresponding record in the other file. The link from the headers file record to the articles file also illustrates the use of indexes for sequential access – having found the first article record for a particular magazine, you could then read sequentially through the rest of them, producing a printed report or screen display.